1987

Sorting with Efficient Use of Special-Purpose Sorters

Mikhail J. Atallah
Purdue University, mja@cs.purdue.edu

Greg N. Frederickson
Purdue University, gnf@cs.purdue.edu

S. Rao Kosaraju

Report Number:
87-695

SORTING WITH EFFICIENT USE OF SPECIAL-PURPOSE SORTERS

Mikhail J. Atallah
Greg N. Frederickson
S. Rao Kosaraju

CSD-TR-695
July 1987

Sorting With Efficient Use of Special-Purpose Sorters

Mikhail J. Atallah[1]

Greg N. Frederickson[2]

S. Rao Kosaraju[3]

Dept. of Computer Science
Purdue University
West Lafayette, IN 47907

Abstract. The following problem was considered by Mueller [M]: Suppose we have a special-purpose hardware device that enables us to sort p numbers at a time, in time $O(p)$. How can a conventional random access machine use such a device to sort n numbers, $n>p$, in better than $O(n\log n)$ time? Clearly, it is impossible to hope for better than $O(n\log n/\log p)$ time performance because of the $\Omega(n\log n)$ lower bound on sorting. Mueller shows that the $O(n\log n/\log p)$ time bound can be achieved probabilistically so long as $p=\Omega(\log^2 n)$. In this note we show that the $O(n\log n/\log p)$ time bound can be achieved deterministically for all values of p.

1 This research was supported by the Office of Naval Research under Grants N00014-84-K-0502 and N00014-86-K-0689, and the National Science Foundation under Grant DCR-8451393, with matching funds from AT&T.

2 This research was supported by the Office of Naval Research under Grant N00014-86-K-0689, and the National Science Foundation under Grant DCR-8320124.

3 This research was supported by the National Science Foundation under Grant DCR-856361.

1. Introduction

Suppose we have a special-purpose hardware device that enables us to sort p numbers at a time, in time $O(p)$. Such devices do exist, e.g., a linear array of p processors. We call such a device a p-sorter. The p-sorter is attached to a conventional random access machine (RAM) that uses its services whenever it wishes to sort p (or fewer) numbers at a time. Each usage of the p-sorter costs the RAM $O(p)$ time steps even if it used it to sort fewer than p numbers (we imagine that enough dummy keys are introduced to fill the sorter). For $n > p$, can the RAM use the p-sorter to sort n numbers in better than $O(n \log n)$ time? Clearly, it is impossible to hope for better than $O(n \log n / \log p)$ time performance because of the $\Omega(n \log n)$ lower bound on sorting.

Mueller [M] shows that the $O(n \log n / \log p)$ time bound can be achieved probabilistically so long as $p = \Omega(\log^2 n)$. In this note we show that the $O(n \log n / \log p)$ time bound can be achieved deterministically for all values of p. We actually give two ways of achieving this bound: one is presented in Section 2, the other in Section 3. The method of Section 2 is based on judicious use of the selection methods of Frederickson and Johnson [FJ]. The second method, given in Section 3, uses a known data structure (the B-tree) in a novel way: elements that are inserted do not travel down to their appropriate leaf immediately and instead "percolate" down the tree rather slowly.

Throughout the paper, all logarithms are to the base two, unless otherwise specified. For simplicity, we assume that the numbers to be sorted are distinct.

2. A Selection-Based Approach

The method we use in this section is based on the following result from [FJ]. In an $a \times b$ matrix ($a \geq b$) whose columns are sorted, selecting the bth smallest element in the matrix can be done in $O(b)$ time [FJ], if the matrix is already in memory, or if any element of the matrix can be produced in constant time. In this section we give a recursive procedure that uses the selection procedure in [FJ] and a p-sorter to sort n numbers. To simplify the exposition, we assume that

$n \geq p$ is a power of p.

If $n=p$ then sort the numbers in $O(p)$ by using the p-sorter once. Otherwise, partition the set S of n numbers into p subsets of size n/p each, and recursively sort each subset. Let S_i denote the ith sorted subset. We select the pth, $(2p)$th, $\cdots$, nth smallest numbers in S in a fashion that we describe shortly. Given the (jp)th element of S, $j = 1,2,\cdots,n/p$, we identify the elements greater than the $(j-1)p$th element and less than or equal to the (jp)th element. These are sorted using the p-sorter, and accumulated in a (column) array of length n.

To select the (jp)th elements and identify the corresponding group of p elements, we do the following. Treat each sorted subset S_i as a column in an $(n/p)\times p$ matrix. Pad each of the sorted subsets S_i with p very large numbers. This ensures that if any number of elements originally in one of the S_i are removed, then at least p elements will remain in S_i. Repeat the following steps n/p times. Select the pth smallest element in S, using the algorithm in [FJ]. Because the contents of each S_i are sorted, [FJ] implies that this selection takes $O(p)$ time. Identifying a group of p elements involves extracting from each S_i (and hence, implicitly, from S) all the elements that are less than or equal to the pth smallest element. Note that the remaining values in S need not be copied to new locations. Instead the index of the beginning of the subsets S_i need only be changed. It does not matter that the columns as handled by the algorithm may no longer be the same length, since the algorithm in [FJ] does not examine any element beyond the pth smallest in any current column.

Correctness of the algorithm follows from the above observations.

Theorem 1. Sorting n numbers recursively using a p-sorter can be done in $O(n \log n/\log p)$ time.

Proof. Each selection using the algorithm from [FJ] will take $O(p)$ time. Extracting a batch of p numbers will also take $O(p)$ time. Sorting these p numbers using the p-sorter again takes $O(p)$ time. These operations will be performed n/p times. Thus the time to combine the sorted

subsets S_i into one sorted list will be $O(n)$. The time complexity $T(n)$ of the algorithm thus satisfies the recurrence

$$T(p) \leq c_1 \cdot p$$

$$T(n) \leq p \cdot T(n/p) + c_2 \cdot n \qquad \text{for } n > p$$

where c_1 and c_2 are constants. This implies that $T(n) = O(n \log n / \log p)$. □

3. A B-Tree Approach

We use a search structure based on a B-tree [BM]. Recall that a B-tree of order m is a tree that satisfies the following properties. Every node has at most m children. Every node except for the root and the leaves has at least $m/2$ children. The root has at least two children, unless it is a leaf. All leaves appear on the same level, and carry no information. A nonleaf node with k children contains $k-1$ keys, which we call search keys.

When a key is inserted into a B-tree, its insertion position is determined relative to the keys in the tree. Thus insertion into a tree with n keys requires $\Theta(\log n)$ time. A sequence of n insertions into an initially empty tree will take $\Theta(n \log n)$ time. If we choose $m = 2p+2$, then the height of the B-tree will be $O(\log_p n)$, which is $O(\log n / \log p)$. This means that $\Theta(\log p)$ time is required in the worst case to determine to which child of a node an insertion value should be sent.

To avoid this cost, we delay sending an insertion value from a node to the appropriate child, until there is a batch of p such keys, which we handle together. We then can use the p-sorter to sort these keys in $O(p)$ time, and then use a merge-like approach to scan the list of search keys and the list of just sorted keys, to determine to which child each key should be sent. The time to send p keys from a node to its children is then $O(p)$ total, or an amortized charge of $O(1)$ per key.

Our variant of a B-tree will have a *bucket* at each node (including the leaf nodes). The

keys in the bucket of a node are keys which would be inserted into the subtree rooted at the node, but for which the insertions have been delayed. The bucket will be organized as a list of keys in unsorted order. We call this variant of a B-tree a *bucket B-tree*.

To insert a new key into the bucket B-tree, insert the key into the bucket of the root. Whenever there are at least p keys in the bucket of any node, extract p of them to form a batch. Sort the keys in the batch using the p-sorter. If the node is not a leaf, perform a merge-like operation with the list of search keys in the node, to determine to which child each key in the batch should be sent. Sending a key to a child involves placing the key in the bucket of the child. If the node is a leaf, then pull the sorted list of batch keys into the parent as search keys. Sort the remaining bucket keys of the leaf, using the p-sorter, and distribute them to the leaves of the parent.

Suppose that a node receives enough search keys from a child so that it has at least $2p+1$ search keys. Identify a middle search key of the node, split the node into two nodes, and insert the middle search key into its parent. One must also split the bucket keys between the two resulting nodes. This involves a scan through the bucket keys to determine which are less than, and which are greater than the middle search key.

When all keys have been inserted, we flush out the buckets, starting with the root, and proceeding down the tree level by level. At each node, sort the bucket keys with the special purpose sorter, at a cost of $O(p)$ time. Then enter each actual key into the bucket list of the appropriate child. If any buckets of children fill up during this procedure, then delay flushing the next node's bucket while these are handled as in the case of an insertion. At the end of the clean-up, a normal B-tree will remain, and a sorted list of its keys can be output in $O(n)$ time.

We observe that the number of bucket keys is well-behaved in the following sense. Let a *breakpoint* of the algorithm be the point immediately after either a new key is inserted in the root, or a batch from one bucket is distributed to its children, or a bucket of a node has been flushed. It can be shown by induction on the number of breakpoints that the following holds. On any path

from the root to a leaf, at most one node has more than $p-1$ keys in its bucket, and this node, if there is one, has at most $2p-1$ keys in its bucket.

Theorem 2. Sorting n keys by using a bucket B-tree and a p-sorter will take $O(n \log n / \log p)$ time.

Proof. We first establish that inserting n keys into an initially empty bucket tree will take a total of $O(n \log n / \log p)$ time. Whenever a bucket key moves from a parent to a child, it is handled as part of a batch of between p and $2p-1$ bucket keys. Sorting these keys, and determining which child each goes to, will take a total of $O(p)$ time, or $O(1)$ per key. A bucket key can move from parent to child $O(\log n / \log p)$ times, until it is a bucket key of a leaf. A total of $O(n/p)$ nodes will be created as a result of n insertions into an initially empty bucket B-tree. Whenever the number of nodes increases during an insertion, it does so as the result of a node splitting. A cost of $O(p)$ is incurred when this happens. Thus the cost resulting from node splits is $O(n)$.

During the flush-out procedure, the time required to flush out partially full buckets will be $O(p)$ total per node, or $O(n)$ total over all nodes. The time to handle buckets that fill during the flush-out procedure may be accounted for as in the paragraph above. □

References

[FJ] G. N. Frederickson and D. B. Johnson, The Complexity of Selection and Ranking in X+Y and Matrices with Sorted Columns, *Journal of Computer and System Sciences 24* (1982) 197–208.

[M] H. Mueller, Sorting Numbers Using Limited Systolic Coprocessors, *Information Processing Letters 24* (1987) 351–354.

[BM] R. Bayer and E. McCreight, Organization and Maintenance of Large Ordered Indexes, *Acta Informatica 1* (1972) 173–189.